Project Integration Architecture: Implementation of the CORBA-Served Application Infrastructure

Dr. William Henry Jones
National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, OH 44135
216-433-5862
William.H.Jones@grc.nasa.gov

X00.00 31 May 2002

Keywords:
PIA; PRICE; CORBA; Network Computing; Application Integration; Application Infrastructure; Distributed User Identification and Authentication; Distributed Access Control Systems; Trusted Server Networks; Collectives; Federations; Encryption; Secured Communications;

ABSTRACT: *The Project Integration Architecture (PIA) has been demonstrated in a single-machine C++ implementation prototype. The architecture is in the process of being migrated to a Common Object Request Broker Architecture (CORBA) implementation and the migration of the Foundation Layer interfaces is fundamentally complete. The implementation of the Application Layer infrastructure for that migration is reported. The Application Layer provides for distributed user identification and authentication, per-user/per-instance access controls, server administration, the formation of mutually-trusting application servers, a server locality protocol, and an ability to search for interface implementations through such trusted server networks.*

1 Introduction

1.1 History

In the late 1980s, the Integrated CFD and Experiments (ICE) project [1, 2] was carried out with the goal of providing a single, graphical user interface (GUI) and data management environment for a variety of computational fluid dynamics (CFD) codes and related experimental data. The intent of the ICE project was to ease the difficulties of interacting with and intermingling these disparate information sources. The project was a success on a research basis; however, on review it was deemed inappropriate, due to various technical limitations, to advance the effort beyond the successes achieved.

A re-engineering of the project was initiated in 1996 [3, 4, 5, 6, 7, 8, 9, 10]. The effort was first renamed Portable, Redesigned Integrated CFD and Experiments (PRICE) and then, as the wide applicability of the concepts came to be appreciated, Project Integration Architecture (PIA). The provision of a GUI as a project product was eliminated and attention was focused upon the application wrapping and integration architecture. During the intervening years, work has proceeded and an operational demonstration of the PIA project in a C++, single-machine implementation has been achieved. This demonstration includes the integration of a Computer Aided Design (CAD) geometry-wrapping application with a wrapped CFD code and the automatic propagation of geometry information from one to the other [5].

1.2 Key Contributions

The PIA technology provides a number of benefits. Among the more significant are the following.

1. Complete engineering process capture is possible to the extent desired.
    (a) A complete derivational history of every project configuration investigated can be captured, producing an auditable trail from final design back to initial guess.
    (b) Technologist's journals, notes, and the like can be captured, allowing the record of thinking to be retrievable in the context of the hard data of the project.

2. Integration of applications into a functional whole is possible, allowing for the complex analysis of entire systems.

3. Rigourous design configuration synchronization is enforced, eliminating mis-matched analyses between integrated applications.

4. The classic *n-squared* integration problem is solved through the use of semantically-defined parameters.

5. Dimensional unit confusion is eliminated by encapsulating in parameters a self-knowledge of their own dimensionality.

6. Quality values (good, bad, and, potentially, a range in between) are captured allowing bad data or designs to be retained in the record without concern that they might be inadvertantly relied upon as being good.

7. Application integration is achieved without the necessity of re-coding those applications to the standard. The wrapping nature of the architecture decouples commitment to the integration standard from the capital assets of the wrapped applications.

8. The wrapping nature of the architecture also allows for multiple wrappers to the same application. Among other things, wrappers appropriate to the skill level of various users might be developed.

9. The architecture provides a significant step forward into the long desired utopia of plug-and-play, mix-and-match software building blocks, allowing customers to pick the analysis pieces needed for a particular situation and drop them into an self-integrating analysis system.

10. The architecture also provides a beginning for the building of intelligence into applications (or more correctly, their wrappers) whereby those applications can search for other applications developing the kinds of information they need. A small peak over the horizon at self-organizing solutions may be here, perhaps the basis for the implementation of the "solve yourself" method.

11. The CORBA-served implementation of the architecture will allow the services of applications to be provided to customers without the release of the actual application software. Often it is the software and its internal techologies which are the competitive edge of an enterprise.

12. Applications made available through the CORBA-served implementation will be more easily maintained since only the copies in execution on the server(s) (as opposed to all the copies that would have been shipped to customers under conventional distribution mechanisms) need be updated when new features are added or mistakes corrected.

2 Developmental Foundations

Before proceeding to discuss the developed application-layer infrastructure, it is appropriate to understand the suppositions upon which that infrastructure is based.

2.1 Commercial, Off-the-Shelf Solutions

One of the points the astute will notice is that the application-layer infrastructure, at points, tends to duplicate facilities and capabilities that might also be obtain from various commerical products or other implementations of standards that are either in place or nearly at hand. In particular, the implementation of object access controls is a topic that has been addressed by the Object Management Group (OMG, the organization responsible for the Common Object Request Broker Architecture (CORBA) standard) and commercial products implementing those further standards are available. Other such areas undoubtedly exist.

Such Commercial, Off-the-Shelf (COTS) solutions are generally declined by the PIA project in favor of project-generated and open-source freeware solutions. Some of the reasons forming this choice are as follows.

1. The technical approaches of some alternative solutions do not fit the (sometimes implicitly understood) design strategies of the PIA project. For example, the commercial products offering object access controls tend toward a single user authentication database server, which represents a single point of failure and runs contrary to the utterly-distributed design goal of the PIA project.

2. As a (possible) standard in its own right, the PIA project is reluctant to build upon COTS solutions as elements of the PIA whole. While the ideal proposes that every COTS solution conforming to a given standard is to be interchangable with every other, the practical reality is that there are often subtle differences between solutions that have slipped between the cracks of the associated standards process. This condition then leads to one of two undersirable conditions: either the PIA implementation would become dependent upon a particular set of COTS solutions to the exclusion of other supposedly identical products, or the PIA implementation would become a morass

of conditional code attempting to accommodate the slight differences between those supposedly identical products.

3. Continuing the previous concern, the Government of the United States is customarily reluctant to provide an endorsment of a particular commercial product in preference to another. Thus, to issue the PIA product with a list of required COTS elements is a less than completely desirable choice.

4. One potential commercialization plan for the PIA project [11] is to provide it as open-source freeware. Such an approach to commercialization offers a number of benefits, including but not limited to wide distribution due to the ease of acquisition, wide acceptance due to the lack of proprietary protections and ploys, confidence due to the ability to examine the exact operations of the software, and a large *de facto* testing/debugging community. Dependence upon a list of required COTS foundation elements is somewhat antithetical to such a commercialization path.

2.2 Implementation Foundation

Except for a few peripheral elements whose need has not yet been clearly defined or established, the implementation of the foundation layer upon which the application infrastructure is built is complete. There are in this foundation layer a few assumptions which run somewhat counter to those of the application infrastructure. The following choices led to this situation.

1. The goal of the foundation layer was to provide generic, reusable structural forms as CORBA-served interfaces. These include arrays, matricies, lists, maps, organizations, graphs, and a few fundamental, non-atomic data types.

2. As a generic, reusable form, the foundation layer does not introduce the concept of a “user”; however, because of the inherently multi-accessor nature of CORBA, resolution of concurrent access conflicts is provided [7].

3. Lacking the concept of a user, the foundation layer had little upon which to hang the concept of an “operating system”. Thus, the central, server control interface (of which each server program has one instance) had little to do but control diagnostic features and provide a skeletal framework for the startup and shutdown of the server program.

4. Lacking the concept of an operating system, the focus of the “bootstrap” process (whereby some initial grasp of CORBA-served objects is obtained) moved to the interfaces themselves. Thus, the **GInterfaceInfo** instances (in CORBA parlance, the interface factories) which parallel the **PClassInfo** objects of the C++ Foundation Classes became the elements publicized for clients to find. Thus, a client does not find a server, but instead finds a served **GInterfaceInfo** instance.

5. Because of an anticipation that there would be many PIA servers in a given, cooperating environment, the publication of **GInterfaceInfo** instances was organized in a way so as to allow many instances supporting the same interface to co-exist. Further, this mechanism allowed the client an opportunity to navigate to a particular **GInterfaceInfo** instance based upon apparent co-locality on the network.

3 Application Infrastructure

As discussed in the following subsections, two key concepts, that of trusted, cooperating servers and that of a “user” combine to shape the application infrastructure.

3.1 The Trusted Server Concept: The Collective

The PIA application model conceives not simply of a single, isolated CORBA server serving one or more compliant application wrappers, but instead a cooperating community of application servers (called a **collective**, so as to avoid calling it a **federation**) serving many different types of applications. The collective may span just a few servers of a division of some corporate entity, or it may consist of thousands of servers cooperating in a world-wide service of technical resources.

Because the collective is conceived as potentially expanding to a world-wide basis, the first design dictum becomes one of scalability through distribution. To the extent that it is possible, there is to be no central resource for anything; no central user identification data base, no central lock or access control mechanism, no central security mechanism or resource, no central location service, etc. This dictum coordinates with the previously explained prediposition against COTS software solutions since many of those solutions gravitate in entirely the opposite direction to central resources and central points of failure.

A consequent point that arises from the distributed, scalable design dictum is that, since many of the activities carried on between cooperating servers involve key security issues, a level of trust between the servers of a collective must be meet. For example, one of the activities carried out by the collective is the identification of a particular user. Since this identification is key to establishing access privileges to information, the validity of the identification process must be maintained. A rogue server accepted by the collective could easily identify its own users as uniquely privileged and, thereby, compromise any and all information throughout the collective.

It should be further noted that the establishment of trust does not directly dictate the open or closed nature of the collective. A collective fully rooted in trust of its members may well permit wide-ranging anonymous or guest account access. The focus of the trust issue is on whether or not the prescribed mechanisms are operating correctly, not the use to which those mechanisms are being put.

At present, no automatic mechanism for the establishment of trust between servers is implemented, or even particularly conceived. Instead, some particular user (typically a server administrator of one or the other servers involved) must be granted ownership privileges to both members of a trusting pair. That user can then execute functionality that forms a linkage of trust between those two servers. It is up to the people involved to satisfy themselves that the requisite compliance to PIA application server protocols is in place on both servers of the established pair.

As additional trust linkages are formed, a graph of trusted servers evolves. Since linkages are reciprocal and adirectional (a linkage indicating the trust of **A** in **B** is matched by a linkage indicating the trust of **B** in **A**, neither of which is considered a "forward" linkage), cycles in this graph are inevitable. Cycles are, of course, not limited to the formed reciprocal linkages. Server **A** might trust **B**, which would trust **C**, which would trust **D**, while **A** might also trust **D** directly.

No initial node of the graph is defined. Because of this, any conceptualization of depth is only relative to the node from which its calculation is started. While technically possible, relative depth within the graph is not presently used to establish any measure of relative trust. Thus, if **A** trusts **B**, which trusts **C**, which trusts **D**, **D** is as trusted by **A** as is **B**.

3.2 The User Concept

Because the application layer is to present for use valuable, indeed often revenue generating, resources, the concept of a "user" which may be granted or denied use of these valuable facilities is very definitely included. This, of course, runs entirely counter to the suppositions of the foundation layer upon which the application infrastructure is built.

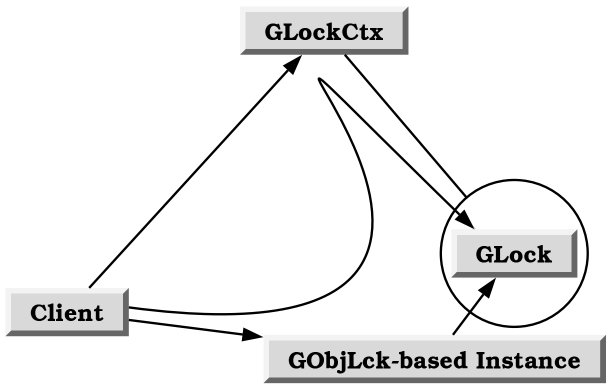

Figure 3.1: Relationship of Concurrency Resolution Components

In the PIA formulation, a user currently has the following rather limited characteristics.

1. A user has a name. This corresponds to the classic computer system concept of the user account, but in the PIA formulation, that name or account is a global concept spanning (potentially) many computers and many PIA-based servers.

2. A user has a location, or more generally, a range of locations from which she originates. In classic computer systems, a user has only one location, the computer system to which she is logged on, and that location is more a point of termination that origination. In the PIA formulation, the point of origin serves to differentiate users in the event of name collisions. Thus, PIA can accept multiple users named "xyz" if those users originate from different ranges.

3.2.1 The User Context Mechanism

The reason for the concept of a user is, as previously stated, to provide a basis upon which to grant or deny usage of the served resources, those resources being in the form of instances of PIA-defined, CORBA-served interfaces. Some mechanism is needed to identify a particular user requesting such resources and to track the resources to which that user has been granted access. The foundation layer provides the basis for such a mechanism in the form of the **GLockCtx** lock context interface.

In the foundation layer, the **GLockCtx** interface is one part of the concurrency conflict resolution triad (illustrated in Figure 3.1) of context (the **GLockCtx** instance), target (any **GObjLck**-based instance), and lock (a **GLock** instance associated with the target) [7]. The **GLockCtx** context tracks the locks currently held by the logical thread of execution. It is presumed that the activities of such a thread of execution are free to perform the acts granted by the locks it holds without further concern of corruption through other concurrent events.

This concept of a context extends naturally into the concept of a user since a user holds the right to access various resources in the form of interface instances. Thus, a derivative form of the **GLockCtx** interface, the **GacLockCtx** interface, is provided to accommodate certain additional functionality to be discussed later.

3.2.2 Provision of Access Control Levels

The next step of the user concept is the recognition that the goal is to provide controls on the access to interface instances beyond the simple resolution of concurrency conflicts. That is, beyond the resolution of whether or not a particular access could be accomplished without corruption is the issue of whether or not a particular user has the right to exercise such an access. This task falls very naturally upon the **GLock** interface which, in the concurrency resolution triad of the foundation layer, makes the decision as to whether or not to grant a lock and, thereby, permit the proposed access. As with the lock context interface, the application infrastructure implementation provides a derivative of the **GLock** interface, the **GacLock** interface, to deal with these matters.

The application layer begins by recognizing the standard access forms defined by the foundation layer: **release**, **reference**, **read**, **write**, **execute**, and **delete**. These are extended into the concept of the right to perform the access, in addition to the present ability to conduct such an access. The application layer then extends these concepts to include additional forms of access, the significant ones being **control**, **own**, and **security**.

Because the implementation is achieved through derivative forms of the **GLockCtx** and **GLock** interfaces, the new access control concepts must be treated in the manner of the old concurrency forms. A certain complexity arises since the new forms enforce the same concurrency restrictions as the old, but must be treated as distinct by the **GLockCtx** and **GLock** mechanisms. For example, both the **write** and **control** access levels require exclusive access to the target instance; however, to hold a **write** lock is not to hold **control** privileges, even though both locks are of equal precedence.

Further, while this distinction between locks of equal precedence but differing privilege is being maintained, it is also necessary to conform to the concurrency system principal that a nested lock application not reduce the precedence of a lock already held. For example should a holder of a **delete** lock request a **control** lock in some nested part of the overall operation, the granting of the **control** lock (which, technically carries only **write** precedence) should not reduce the **delete** precedence already held by the context. All of this must be done, of course, by code that has no knowledge that derivative lock/privilege forms have been defined.

The implementations of the **GLockCtx** and **GLock** interfaces provide a few functional hooks that allow this introduction of additional lock/privilege concepts. The **GsLockCtx** class (which implements the **GLockCtx** interface) provides the **IsLesserLock** member function which compares the precedence of two lock levels. This function, in turn, uses the **ConvertToLockLevel** method of the **GLock** interface to obtain a basic lock precedence for a provided lock level. The implementation of the derivative **GacLock** lock interface provides an override of the defined **ConvertToLockLevel** function that adds the knowledge of the newly-defined lock/privilege levels and provides the appropriate foundation level precedences for each. In the event that a precedence adjustment must be made, the **GacLock** interface supplies an override of another method, **PromoteToPrecedence**, which supplies lock level codes for the desired access kind at the required precedence. For example, if the current precedence is **execute** and a **control** lock is requested, the overridden **ConvertToPrecedence** method implementation provides a code that is, in its essense, **control at execute precedence**.

3.2.3 Enforcement of Access Control Levels

Having, through object-oriented slights of hand, introduced additional access control levels and kept them distinct in the implemented concurrency mechanism, the next magical feat is to actually enforce those levels. This is done through an override of the implementation of the **RequestLock** functionality originally defined by the **GLock** interface. The overriding code is provided by the implementation of the derivative **GacLock** interface.

The **RequestLock** override eventually relies upon its inherited base-class implementation to provide concurrency resolution; however, before it does so it first verifies that the user represented by the supplied **GLockCtx** interface (which must be, in fact, a **GacLockCtx** interface), can exercise the privileges of the requested lock. The source of this answer is found in a **GacDescAccs** access control descriptive element attached to the lockable target, which itself must be an instance of the **GacBObj** interface (or its derivatives). (This symbiosis between **GacBObj** targets and **GacLock** locks is enforced through an override provided by the **GacBObj** implementation of the **NewLockInstance** member function originally defined by the implementation of the **GObjLck** interface, **GacBObj** being a derivative of **GObjLck**.)

The **RequestLock** override obtains from its supporting **GacSrvrCtl** server control instance (the **GacSrvrCtl** interface being a derivative of the **GSrvrCtl** interface of the foundation layer) a user identification text associated with the **GacLockCtx** instance requesting the lock. The descriptive sets of the particular **GacBObj** interface are then searched for a **GacDescAccs** instance providing a definite answer as to the access privileges of the identified user. If no such answer is found in the course of the search, the access is, currently, denied.

Note that the **GacBObj** descriptive system is hierarchial in nature, providing many potential layers of description corresponding to each layer of derivation from the **GacBObj** foundation. Each such layer may provide a separate **GacDescAccs** access control descriptive instance. The layers are searched from shallowest (the most derived layer) to deepest (the **GacBObj** layer) while the issue remains in doubt.

The access control search further recognizes the application layer concept of structural uplink and, while an answer to the access control question is not yet found, will proceed up this chain of **GacBObj** instances to containing logical application structures. The first instance providing a definite answer terminates the search and settles the issue. Because of this upward search, the potential exists for controlling many instances through a highly-placed access control description. For example, access controls might be placed on a root application instance (that is, an instance of a derivative of **GacAppl**) and neglected on all the components of the application that instance heads. Because of the uplink search protocol, every component ever added to that application instance would be governed by the access controls of that single **GacAppl**-derivative instance.

3.2.4 Access Control Description

The **GacDescAccs** access control interface does not, itself, implement the access controls. Instead, it serves only as a linkage to a more general set of control mechanisms, the **GacAccsCtrl** access control interface which organizes **GacAccsAce** access control entry instances. **GacAccsAce** instances actually record the particulars of privilege granted to a specific user or account.

As currently implemented, **GacAccsAce** access control entries (ACEs) are organized by **GacAccsCtrl**-derivative forms into access control lists (ACLs) which are traversed from head to tail. Again, the first such entry providing a definite answer terminates the access control search and determines the issue.

ACEs identify users either by a simple text match of a provided text, or by the matching of a general regular expression to that text. By using a general regular expression, a user may be allowed to roam over a range of machines while still exercising the same access privileges. Additionally, the list may be arranged as a filter through the use of multiple entries applicable to a partiular user. For example, such a list might grant greater privilege to a user when that user originates from a more restricted (and, presumably, more trusted) range of machines.

3.2.5 Access Control Execution: The Privileged Account

The descriptive system interfaces used to describe the access controls of **GacBObj**-derived interfaces are, of course, themselves derivatives of the **GacBObj** interface. The access control descriptions thus avail themselves of the same access control protections as the instances they, themselves, protect. That is, to read an access control description to determine if it grants some particular access to an identified user, one must obtain a **read** lock on that descriptive instance. And to obtain that **read** lock, the **GacLock** instance protecting the access control description must determine that the requesting **GacLockCtx** can exercise **read** privileges on that instance. Without some relief, an infinite recursion immediately develops: to gain access to a target, a requestor must gain accesss to another target, but to gain access to that target, the requestor must gain access to another target, and so on.

To break this recursion, the determination of access privileges is not carried out in the context of the requesting **GacLockCtx** instance. Instead, the **GacLock** instance goes, once again, to its supporting **GacSrvrCtl** server control instance to obtain a privileged lock context in which to conduct the access control search. When a **GacLock** instance requests a user identification of a privileged lock context, the special identification "server_root" is returned. The **RequestLock** override of the **GacLock** implementation is coded to recognize this special identification and bypass the entire access control process when it occurs. Naturally, considerable pains are taken to assure that the identification "server_root" is not granted to other lock contexts. Even system administration accounts, which customarily have full access privileges, are named "server_admin" rather than “server_root” as a security precaution.

3.2.6 Control of Access Controls

The modification of access controls is not carried out in a privileged context, but instead must be the act of a user supplying a **GacLockCtx** instance. Since, as previously pointed out, the access control interfaces are themselves derivatives of the **GacBObj** interface, they too can attach access controls for their protection. The access control recursion problem is, thus, re-encountered. Two solutions are available.

First, an access control could attach no access controls and rely upon the uplink search protocol to define its accessibility. Since the uplink of an access control proceeds up to the **GacBObj**-derivative instance it controls, by waiting for this mechanism to operate, an access control would become self-controlled; access privileges to the control become identical to access privileges to the controlled instance.

As a second alternative, a self-controlled characteristic can be set in an access control which achieves the same result: access privileges to the control are identical to those of the controlled instance. This alternative is provided for those situations in which the uplink search protocol is not available, or when it is desirable to circumvent that protocol.

3.2.7 Concurrency Resolution

Once an affirmative determination of privilege is obtained, the requested lock is converted to a foundation layer code and the inherited **RequestLock** functionality of the **GLock** implementation is invoked. This functionality proceeds in the usual manner to resolve conflicts of concurrent access and either grant or deny the requested lock.

3.3 Association of Identities with Lock Contexts

As mentioned above, it is the job of the **GacSrvrCtl** server control instance supporting a PIA-compliant application level server to associate a user identification (in the form of a text) with a particular **GacLockCtx** instance. At present, this is done in the following manner, which is illustrated in Figure 3.2.

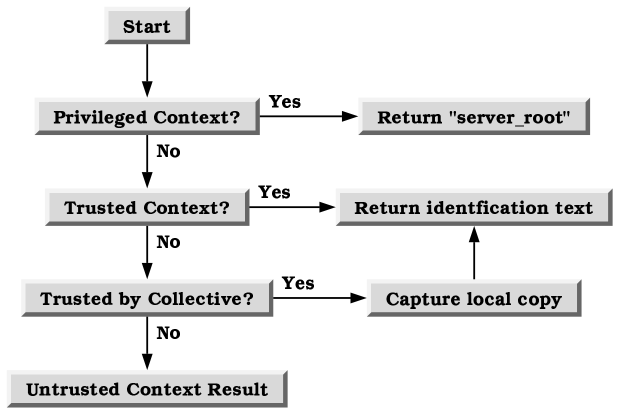

Figure 3.2: Association of Identity with a Lock Context

1. The instance name of the identified **GacLockCtx** instance is obtained.

2. An internal, **PMap**-based map is searched for that instance name. If that name is found and a further Interoperable Object Reference (IOR)-based test is passed, then the identified **GacLockCtx** instance is one of the privileged lock context instances maintained by the **GacSrvrCtl** instance and the special identification “server_root” is returned.

3. An external, **GMap**-based map is searched for the instance name. If the name is found and, again, a further IOR-based test is passed, then the identified **GacLockCtx** instance is an instance already known to the **GacSrvrCtl** instance. A linkage to a **GacTrustedLcxInfo** instance is followed, a user identification text obtained from that instance, and that text returned as the associated identification.

4. Should the identified **GacLockCtx** instance still be unrecognized, a traversal of the other servers of the collective is conducted in the expectation that some server will recognize the context. Should some server respond positively, that response is honored as being valid and the subsequently obtained identification passed back to the inquiring **GacLock** instance. The association is also recorded in the **GacSrvrCtl** instance hosting the original inquiry in the expectation that the issue will arise again.

5. Should the **GacLockCtx** instance still be unrecognized, an empty user identification text is returned. The empty text is defined as being a declaration that the **GacLockCtx** instance is not trusted. Operations in such a context should be refused.

It is in the traversal of the collective step that the issue of trust between members can be clearly seen. A rogue server can easily answer that the associated identification is the all-powerful “server_root” pseudo-user. Such a response cannot be rejected *a priori* since operations in a privileged context are not prohibited from crossing server boundaries.

That is, there are legitimate situations in which a server may encounter the privileged lock context of another member of the collective. If members of a collective were simply to trust any other server that they might detect, the breaching of PIA application infrastructure security would be trivial.

Note that the mechanism described above conforms to the distribution of services design dictum established for the PIA application infrastructure. There is no central resource for establishing the user association of a particular **GacLockCtx** instance. Instead, it is a distributed act of the collective operating under the presumption that somewhere "out there" a member exists that can identify the lock context instance and associate a user with it. Furthermore, the operation is conducted in a manner so as to tolerate the occassional unavailability of some trusted servers.

3.3.1 Lock Context Linkages

As indicated previously, each **GacSrvrCtl** server control instance keeps a map of **GacLockCtx** instances which it has, by one means or another, identified as being trustworthy.

Since it is anticipated that the lifetime of a **GacLockCtx** lock context instance will correspond with the working sessions of the user it represents, it is necessary to provide a mechanism to notify each trusting **GacSrvrCtl** instance when a trusted **GacLockCtx** instance becomes **defunct** (that is, when the instance is destroyed). Thus, the **GacLockCtx** derivative implementation has been coded to include a map of **GacSrvrCtl** instances trusting the particular **GacLockCtx** instance. An entry in this trusting server map is made at the time the **GacSrvrCtl** instance adds the **GacLockCtx** instance to its trusted lock context map. (This is one of the reasons that all lock contexts used in PIA-compliant application servers must be of the derivative form **GacLockCtx** rather than the base **GLockCtx** form.) When the particular **GacLockCtx** instance becomes **defunct** at the end of a user session, it traverses this map and notifies each identified **GacSrvrCtl** instance so that the trust linkages may be discarded.

It is further anticipated that the lifetime of a server (and, consequently, its associated **GacSrvrCtl** instance) will be far longer than that of the **GacLockCtx** instances it trusts, extending out from months and years toward a practical infinity. Thus, the need for the reciprocal mechanism for disolving linkages of trust to such **GacLockCtx** instances is much smaller. None the less, the mechanism is provided for the dissolution of linkages upon the demise of a server.

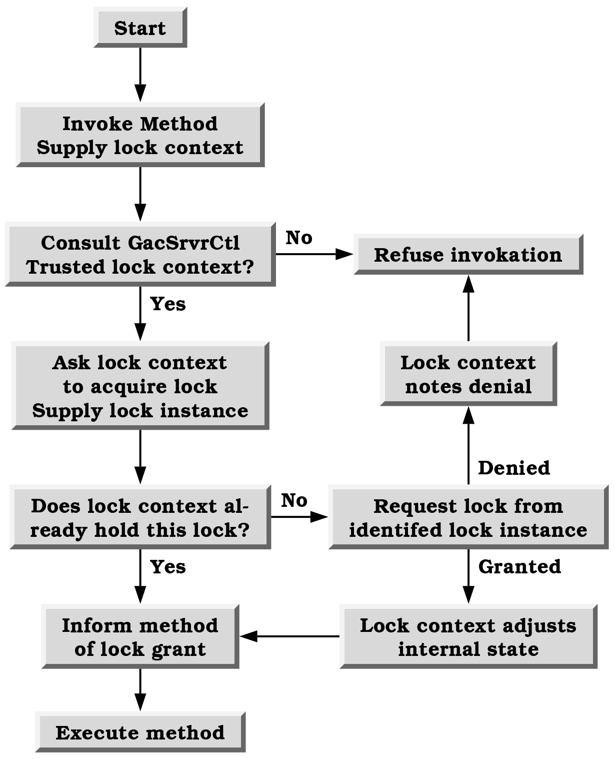

Figure 3.3: Flow of Events in Lock/Access Acquisition

3.4 Establishing Trust in Cooperating Components

Based upon the system described in [7], the flow of events in acquiring a lock (and, with the introduction of the application infrastructure, access privileges) is in the following manner, as illustrated in Figure 3.3.

1. The user invokes a method on the target (**GObjLck**-derivative) instance. A **GLockCtx** lock context instance (which is, in fact, of the derivative kind **GacLockCtx**) is supplied as the user's context for the operation.

2. The target instance consults with its associated **GacSrvrCtl** instance to determine if the supplied **GLockCtx** instance is trustworthy. If this is not the case, the method execution is aborted with a lock fault indication.

   This step, introduced by the application infrastructure, is a regrettable overhead since it must be performed even when the supplied lock context instance already holds the desired lock; however, the step is necessary to prevent the introduction of rogue lock contexts that will return false results to the target instance. The overhead is even greater than is apparent because of the often recursive nature of method implementation:

method **A** obtains a lock and then invokes method **B** on the same instance, which obtains a lock and invokes method **C** on the same instance, which obtains a lock and....

To alleviate this situation, the **GacBObj** application foundation interface implements a small, hashed, associative cache of lock context instances recently demonstrated as being trustworthy. Concurrency resolution of this cache is by means of mutual-exclusion locks executed on a per-element basis; thus, maximum execution concurrency is expected. It is expected that the storage and execution burden of this cache will be far outweighed by the overall performance improvement gained.

3. The target instance requests that the supplied (and now trusted) **GLockCtx** lock context instance obtain a lock of appropriate kind on the target instance. A reference to the correct **GLock** instance (which is, in fact, of the derivative kind **GacLock**) is supplied in that request.

4. Assuming that the lock context instance does not already hold the requisite lock, it makes a further lock request on the identified **GLock** instance.

5. The **GLock** instance evaluates the request and either grants or denies the requested lock, returning that result to the requesting **GLockCtx** instance.

6. The **GLockCtx** instance makes such notations as are appropriate to the result of its request and then returns its lock request result, again usually of the form grant or deny, to the requesting target.

7. The target instance examines the result returned to it and, if the required lock is granted, proceeds to perform whatever operation the invoked method encapsulated.

The security of all this depends, of course, upon each of the three components, target, context, and lock, performing as they are designed. The security of two of the components, the target and the lock, is not in doubt in the application infrastructure.

1. The target, **GObjLck**-derivative instance is considered to be secure by definition. There can be no point to implementing an inherently corrupt target instance since the serving of such an interface would be pointless.

2. The security of the lock is entirely controlled by the target instance. The lock instance is obtained through an internal member function of the target interface implementation. There is no mechanism for attaching an alternative, corrupt lock instance to a target.

Unfortunately, the third element of this triad, the **GLockCtx** lock context instance, is entirely amenable to security breaches through the introduction of a corrupt instance. Throwing aside all sorts of devious mechanisms, a corrupt **GLockCtx** instance may obtain all the access it desires simply by returning a grant result to the target without regard to the actual result returned to it by the **GLock** lock instance. (Indeed, why even bother inquiring of the lock instance?) Further, the location transparency features of CORBA create a situation in which an individual intent upon breaching security is completely able to substitute her own **GacLockCtx** implementation to undertake whatever nefarious scheme she might devise.

Many schemes for validating a supplied **GLockCtx** instance through operational tests were considered; dummy lock operations with known results, preset operations on the locks of the intended targets, location of the serving server, and the like. Counters to all such validity tests we identified and, ultimately, the concept of accepting a **GLockCtx** from an unverified source was discarded.

As a result of the above considerations, it was determined that a user would have to go through an initial server logon sequence, the result of which from the user's perspective would be a reference to a **GacLockCtx** instance obtained from and trusted by the providing **GacSrvrCtl** server control instance. It is this logon operation that makes the initial entry of a **GacLockCtx** lock context instance into a trusted lock context map of a server and, consequently, into the collective.

Trust is established in the **GLockCtx** instance because it is supplied from a trusted source. Further, because the server control instance closely tracks through information interior to the server the identity of each **GacLockCtx** instance it issues, it is not possible for the user to substitute a corrupted lock context instance in place of even a legitimately obtained lock context.

The final barrier to rogue lock context substitution is based upon comparison of the IORs for the supplied and trusted instances. At the time a trusted lock context is generated, its IOR is recorded. When a supplied lock context instance is examined, the apparent IOR for that instance is compared with the recorded IOR: if the two differ, then a substitution must have been made.

The use of the IOR for identification is, in general, not in compliance with the CORBA standard. The standard allows a particular instance to be served by different servers at different times. Since the IOR includes the information necessary to locate the presently served instance, a migrating instance exhibits a varying IOR.

The difficulties of the IOR comparison are eliminated by the design assumptions of the PIA effort. PIA instances are to persist until deliberately destroyed, even across server shutdown/restart cycles of any duration. Since references to such persistent instances are maintained by recording the IORs of those instances, the migration of PIA-conformant instances to other servers (with the attendant invalidation of outstanding IORs) is prohibited.

Because of the number of instances anticipated in real-life PIA implementations, no thought of central, forwarding instance registries to bridge the fixed-IOR-to-migrating-instance gap is entertained. Typical instance counts in the range of many billions and up are expected in such real-life implementations.

3.5 The Logon Operation

Like other elements of the PIA application infrastructure, the logon operation is a fully distributed act. A user may log on to any server of a collective without regard to whether or not that particular server contains the user's account information.

The user initiates the logon operation by invoking the **UserLogonRemote** method in presentation by any **GacSrvrCtl** server control instance of the desired collective. In response, the user identification and password are solicited through a supplied **GacUser** user interaction interface. If the particular server control does not recognize the resulting user identification, the collective is searched for a **GacUserInfo** user information instance. Assuming that a member of the collective responds in the affirmative to this search, the obtained logon information is transmitted to the responding **GacSrvrCtl** server control instance and authentication completed.

With the user identification verified, the server control instance initially handling the logon request allocates a trusted **GacLockCtx** instance, enters it into the internal structures tracking such instances, and associates with it the user identification finally established by the logon process. Ultimately, a reference to the trusted **GacLockCtx** instance is returned to the user as the result of the logon process.

Support for both pre- and time- expired passwords is provided. If the logon operation discovers such an expired password, a password change operation must be successfully completed before the logon operation can complete. The password change operation excludes previously-used passwords and can enforce the usual and customary rules for password composition. Again, the actual password is changed and maintained by the **GacSrvrCtl** claiming knowledge of the user.

3.5.1 Administration of User Accounts

The design of this user information system is intended to allow a user's information to be maintained on a single server of convenience while not restricting the points of access available to that user. For example, if a number of corporations have formed a collective of servers, user account information for an employee of a particular corporation can be maintained on a server provided by that corporation without the need for that user to log on to that particular server. This allows the user to obtain the **GacLockCtx** resource on the member of the collective in which she intends to be most active. Because of the high interaction rates of the lock context instance, this may be a performance advantage in some situations.

The logon process does not specify a precedence in the event that more than one **GacUserInfo** instance applicable to a particular user exists within the collective. The first applicable instance found in the traversal of the collective is the instance used; however, it is not predictable which member of the collective will have the first opportunity to respond. Furthermore, the search is not continued to identify additional applicable **GacUserInfo** instances, even though they might exist elsewhere in the collective.

The unpredictability in the case of multiple user information instances is not considered to conflict with the current PIA application infrastructure design. A single **GacUserInfo** user information instance within the collective is considered to simplify such administrative tasks as password management, account disabling and the like. The single instance design does represent a single point of failure since, if the appropriate member of the collective is unavailable, the users supported by that member cannot complete a logon sequence; however, that failure is only for that group of users, not for the users of the collective as a whole.

A multiple **GacUserInfo** instance design is possible and support for such configurations may be implemented in the future if server availability issues warrant such facilities. It is hoped that the ease of single instance administration combined with reasonable server reliability will suffice for the time being.

3.5.2 Organization of Accounts Within a Server

As mention earlier, the collision of account names is expected, especially so in the case of global collectives. To resolve such collisions, the user is also identified by a point of origin. A general regular expression is used to provide a particular user a range over which she is recognized.

Because of this user identification arrangement, it is not possible for a server control to deterministically identify a **GacUserInfo** instance through a single mapping structure. Instead, lists of **GacUserInfo** instances are sorted by their common (collided) user name. Once an appropriate list has been identified, it is traversed from head to tail applying the general regular expression of each encountered **GacUserInfo** instance to the actual location associated with the user. The first match that is found terminates the list traversal and selects the enumerated **GacUserInfo** instance.

While the general intent of this user identification system is that only one **GacUserInfo** instance be applicable to any given user, it is possible to use the system in a filtering manner. For example, very specific ranges for a given user might preceed much more general location ranges in the list, with the effect that the user would be in some way different when originating from the constrained locations. Currently, the user identification system serves only to provide identity and, thus, such distinctions as might be accomplished through such a filtering system are trivial; however, at some future point, useful distinctions such as billing and credit, service priority, and the like might be controllable through such a system.

3.6 Protection of Sensitive Information

The previous section mentioned the transmission of user identification, including passwords, between the **GacUser** user interaction instances and various **GacSrvrCtl** server control instances of the collective. In point of fact, passwords are considered by the PIA application infrastructure to be sensitive over and above the sensitivies of other information. While it is expected that collectives handling sensitive information (whether legally secret or related to the competitive advantage of a business) will routinely deal in secured communications technologies such as Secure Socket Layer (SSL), passwords in particular are protected even within such secured transactions.

Each **GacSrvrCtl** maintains a ready supply of encryption keys using the algorithm of Rivest, Shamir, and Adleman (that is, the RSA algorithm). The PIA application infrastructure defines the acquisition of a keyset with both the public and private elements intact as being a protected function of the **GacSrvrCtl** interface which, generally, is only to be exercised between instances served by the same server so as not to expose the private key content to possible eavesdropping. On the other hand, the provision of a key with only public encryption elements is entirely open and such a key is considered to be freely transmittable between instances without regard to their relative locality.

All derivatives of the **GacBObj** application foundation interface inherit the ability to provide and utilize something called a **passback encryption key** for the purpose of protecting the transmission of sensitive information to an instance of that interface. (Certain other interfaces, in particular the **GacSrvrCtl** interface which is not a derivative of the **GacBObj** interface, implement this functionality by other means.) The cycle of operation is in the following manner.

1. The instance intending to transmit sensitive information requests a passback encryption key from the intended destination instance.
2. The destination instance locates the **GacSrvrCtl** server control instance associated with the server program serving the destination instance and acquires from that server control instance a complete RSA encryption key.

   Because this transaction is entirely interior to the single server program, the transmission of a complete RSA key between instances is considered to be acceptable. The premise is that a server must be able to trust itself.
3. The destination instance records the complete RSA encryption key in an internal structure, sorting the key in that structure by the public modulus of the key.
4. The destination instance then returns to the requesting source instance the public encryption portions of the key.
5. The destination instance encrypts the sensitive information using the received public, passback encryption key and invokes a method of the destination instance. The public, passback encryption key is supplied to the method invokation as one of its arguments.

   Only particular, documented methods of an interface support the passback mechanism, and in those cases only particular arguments are encrypted.
6. The invoked method locates the complete passback encryption key in its internal structure based upon the public modulus obtained from the public key portion supplied as an argument to the method. The sensitive information is decrypted and method execution proceeds.
7. Finally, the destination instance discards the used passback encryption key. Any subsequent passback

encryption operation will require a new encryption key.

In the event that an intended passback encryption operation does not come to pass, mechanisms are provided to return the public portion of the passback encryption key to the intended destination instance so that the key may be removed from the internal structures of that instance. Even in this case, though, the encryption key is discarded. Until the random generation process regenerates the same key again (something thought to be unlikely in the extreme), an eavesdropper will find a gleaned key useless with regard to future operations.

With this understanding of the passback encryption mechanism in hand, it is merely necessary to add that all password transactions are handled in this manner. The **GacSrvrCtl** instance on which the logon operation is begun provides a passback encryption key to the **GacUser** instance conducting the interaction with the user. The password is transmitted to the **GacSrvrCtl** having access to the actual **GacUserInfo** instance using a passback encryption key obtained from that server control instance. The password is then transmitted on to the **GacUserInfo** instance in the same manner.

The **GacUserInfo** interface protects encapsulated passwords by RSA encryption once they have been received. Again, an encryption key is obtained from the associated **GacSrvrCtl**, but in this case only the public portion of the key is requested. Thus, once encrypted for storage by the **GacUserInfo** instance, the original plain-text form of the password is no longer obtainable.

A number of operations enforcing various rules for the form and length of a password must be carried out on the plain-text version of that password. These operations are confined strictly to the internal mechanisms of the **GacUserInfo** interface and, as such, are considered to be sufficiently secure. Note, though, that this has it ramifications: it is presently considered an unacceptable security risk to pass the plain-text password to shared resources such as common prohibitied password dictionaries and the like. Such mechanisms might be arranged and made secure by further employing the passback encryption key mechanism; however, these mechanisms would not be able to use the repetoire of distributed object capabilities provided by the CORBA-served PIA implementation. For example, a **GMapGStr**-based map of prohibited passwords would not be possible because the interface does not support passback encryption. Even if that capacity were added, the computational burden of encryption would be beyond realistic achievement.

3.7 Location of Interfaces

Very nearly the first issue to be confronted by any CORBA client code is how to find a served instance or obtain a new served instance. As mentioned at the beginning, the PIA foundation started this issue off in a direction which must be reversed by the application layer.

3.7.1 The Foundation Layer Approach

The PIA foundation layer provided no concept of a user and, lacking that, had little to define a system. The central **GSrvrCtl** server control instance of a foundation server has little to do but start and stop the server and turn debugging logs on and off.

Because of the lack of user and system concepts, the focus is upon the **GInterfaceInfo** interface and its instances. One instance of this interface is created and served for every interface served by the foundation layer server. The created instance is named for the interface it supports. For example, an instance of the **GInterfaceInfo** interface named **GMapGObjToGObj** is created and served to support the **GMapGObjToGObj** interface. To have a reference to that **GInterfaceInfo** instance is to have the ability to create and use instances of the **GMapGObjToGObj** interface.

As each **GInterfaceInfo** instance is created, a reference to it is published by the foundation layer in a well-known **NameService** server. After several name context layers sorting through the fact that the reference is a PIA-conformant **GInterfaceInfo** instance, the reference is distinguished by its assigned name. To account for the fact that multiple PIA servers serving many (if not all) of the same interfaces are expected to exist, the reference is further qualified in the naming service by appending additional layers consisting of the fully-qualified domain name of the server. The order of the domain name components is reversed so that they proceeded from most general (**.gov**, **.com**, **.org**, and the like) to most specific. Only when the terminal element of the domain name is reached is an actual reference to a **GInterfaceInfo** instance obtained. In this way, a **GInterfaceInfo** instance supporting the **GMapGObjToGObj** interface on one serving machine is distinguished from another **GInterfaceInfo** instance supporting the same **GMapGObjToGObj** interface on the next machine over.

Note should be taken of the fact that the fully-qualified domain names used are established by configuration actions of the PIA-compliant server programs and not by making inquiries of any actual Domain Name System (DNS) server that might be available. While it is generally intended that configured server names will follow the DNS names of the serving machines, the configuration option allows deviations from those names that may serve useful purposes. For example, a group of machines not sharing any particular pattern of DNS names but all serving a common application may be formed into a server cluster by placing a common name suggestive of the served application just before (in reversed order, or after in DNS order) the terminal, machine-identifying name. For example, a cluster serving the LAPIN code might be configured to exhibit the following server names.

```
srvr00.lapincluster.grc.nasa.gov
srvr01.lapincluster.grc.nasa.gov
srvr02.lapincluster.grc.nasa.gov
srvr03.lapincluster.grc.nasa.gov
srvr04.lapincluster.grc.nasa.gov
srvr05.lapincluster.grc.nasa.gov
srvr06.lapincluster.grc.nasa.gov
srvr07.lapincluster.grc.nasa.gov
```

The foundation layer provides services for the navigation of the name service structure it has constructed. In general the service proceeds in the following manner.

1. First, the naming contexts are navigated to the **GInterfaceInfo** instance tree supporting the desired, named interface, for example up to the point where the **GInterfaceInfo** instance(s) supporting the **GMapGObjToGObj** interface is identified.

   Note that, once this navigation phase is completed, it is certain that any **GInterfaceInfo** instance identified will support the desired interface. In the case of a very general interface such as **GMapGObjToGObj** there may yet be many possibilities left. In the case of a very specific interface, for example a (supposed) **LapAppl** LAPIN application wrapper interface, there may be very few possibilities left.

2. From the currently identified point, naming contexts are further selected based upon the (reversed) fully-qualified name of the client.

   Using the example of the supposed LAPIN cluster given above, a client named

   ```
   somemachine.grc.nasa.gov
   ```

   will navigate up through **gov**, **nasa**, and **grc** since those elements match the client name.

   It is this phase of the navigation process that attempts to achieve network co-locality. It is presumed that matching name components will be indicative of "closeness" in some network sense.

3. From the currently identified point, further naming contexts are navigated by random selection until a terminal context is reached. The charitable might consider this last random selection among servers to be a minimal form of load balancing; the uncharitable might not.

   Further using the supposed LAPIN cluster example given above, having reached the naming context of **grc**, random selection has only one choice: **lapincluster** (presuming for the moment that only members of this cluster serve the desired interface). From that point, one last random selection picks a server of that cluster and leads to a final reference to a **GInterfaceInfo** instance supporting the desired interface.

As almost an afterthought, the foundation layer similarly publishes a reference to the **GSrvrCtl** server control instance of each server.

The above implementation represents what could be done within the very general framework assumed for the PIA foundation layer. It should be noted that all of the publication actions are implemented as options which derivative servers may turn off.

3.7.2 Deviation from DNS Names

The example of the supposed LAPIN cluster of servers in the previous section served no particularly spectacular purpose. Since it was presumed that the desired interface was served only by members of that cluster, one of those members would have been selected even if the client name had been

```
somedesktop.bldg666.seattle.bcac.com
```

As a more constructive example (which leaps just a bit forward into application layer concepts), suppose this LAPIN cluster consisted of power server machines on which it was inappropriate to run PIA-conformant GUIs for administering those servers. Let us suppose, instead, that there are several desktop machines named

```
admin00.lapincluster.grc.nasa.gov
admin01.lapincluster.grc.nasa.gov
admin02.lapincluster.grc.nasa.gov
admin03.lapincluster.grc.nasa.gov
```

and existing on networks judged secure enough to perform administrative tasks.

Under this set of suppositions, the default instance location mechanisms will lead these supposed administrative machines straight to the servers they propose to administrate. This will occur even for instances of interfaces served by many other servers outside this cluster.

3.7.3 The Application Layer Approach

Two key points adjust the application layer approach to locating interface services.

1. The **GacSrvrCtl** server control instance is now a vital part of a PIA-compliant application server. In particular, the need to locate such an instance to logon and obtain a trusted **GacLockCtx** lock context instance is paramount.

2. The number of interfaces to be served by PIA-compliant application servers and the number of expected servers combine to make the burden upon supposed **NameService** servers untenable.

   One of the visions of the PIA plan is that every kind of engineering, technical, scientific, management, manufacturing, quality-control, or other parameter will be encapsulated in a specific, closely-defined interface so that its well-known, pre-defined semantics may be recognized by discovering its interface type and functionality specific to those semantics may be encapsulated. This alone may lead to tens or even hundreds of thousands of defined interfaces. When this is multiplied by a supposed global collective, many of whose members serve the same interfaces over and over again, the untenable burden upon even a federated name service becomes clear.

Because of these factors, servers of the application infrastructure turn off the **NameService** publication of **GInterfaceInfo** instances, even though these instances are still created at server startup for every supported interface. Also, while the **GacSrvrCtl** server control instance is still published in the normal, foundation layer manner, the services provided by the collective mean that each member of the collective need not publish their individual server control instances with any one name service. Indeed, some members of the collective need not publish their server control instances at all if that is not desired. The full services of the collective may be reached through a local name service providing connections to only a few, local members of that collective.

Having turned off the general publication of **GInterfaceInfo** instances, the application layer must provide an alternative method of locating those interface-supporting instances, which it does in the form of the **TrustedSrvrFindIifByName** method and the **GacFqdnToGObj** organizational structure.

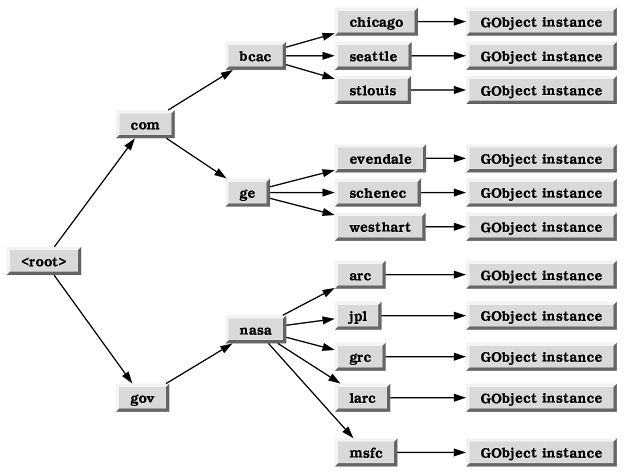

Figure 3.4: Organization of Fully-Qualified Names by the GacFqdnToGObj Interface

As illustrated in Figure 3.4, the **GacFqdnToGObj** interface provides an *n-ary* tree facility used to create a correspondence to (reversed) fully-qualified names. The root instance of the structure customarily has no name while each of the root's immediate offspring are assigned the most general element of the fully-qualified name being used to map a path through the structure. The next element of the fully-qualified name identifies the offspring of those offspring, and so on until terminal nodes are reached. Any node of the constructed structure may provide a reference to an instance of any **GObject**-derivative interface; however, in common use only terminal nodes of the structure have such references.

The **TrustedSrvrFindIifByName** method builds a **GacFqdnToGObj** structure identifying, much in the manner of the foundation layer's utilization of the name service, all of the servers providing a **GInterfaceInfo** instance supporting a named interface. The path through the the **GacFqdnToGObj** structure is navigated/constructed by the fully qualified names of those servers and the terminal node identified provides the reference to the identified **GInterfaceInfo** instance. The operation traverses the collective, identifying every member serving the identified interface. Once the **GacFqdnToGObj** result is received by the requesting client, it can be navigated by fully-qualified name and random selection, in the manner of the naming contexts of the name service, to an appropriate **GInterfaceInfo** instance.

There are good and bad points to the application layer approach to finding an appropriate **GInterfaceInfo** instance.

1. The entire collective is searched for services, without the need of publishing all such services in a single name server reachable by the client. Even collective members that have not published their server control instances anywhere are reached.

2. Only the reference structure for the desired interface is generated, not that of all possible interfaces. In the case of narrowly focused interfaces, for example a (supposed) **LapAppl** application wrapper interface, this is likely to be a much smaller and more manageable result.

3. The reference structure is built within the PIA persistent object model. Should the structure become inordinately large (for example, in inquiring which members of a world-wide collective serve, of all things, the **GObject** interface), less active components of the structure will be etherealized and reincarnated should they be needed again.

4. The reference structure must be (re-)generated for every inquiry.

   This is the only advantage of the foundation layer's name server approach over the application layer's facility. The name server structure is generated once and then need only be navigated when the need arises.

The application layer anticipates the publishing of instances of interfaces derivative of **GacAppl**, the generic application wrapper interface. While not yet implemented, inquiry into published **GacAppl**-derivative instances is expected to be along the lines of the **GInterfaceInfo** mechanism, resulting in a **GacFqdnToGObj** structure that may be navigated as desired. In this case, though, network colocality may not be a desirable discriminant; however, since the **GacFqdnToGObj** interface is derivative of the **GObjDgn** directed graph interface, other traversal options do exist.

Several elements of application wrapper publication seem, at this point, apparent.

1. The **GacSrvrCtl** server control interface will either have to encapsulate a publication structure or maintain a reference to a devised structure.

2. Published application wrapper instances will have to be distinct by name. The given PIA instance name meets this purpose. Additional semantic discrimination may be obtained by directly interrogating the descriptive system of the identified application instance.

3. It would be desirable to apply the principal of semantic infusion through interface derivation to the application interface as has already been done in the prototype work for parameter classes. In this way a client seeking to find published applications in a particular discipline, for example in the discipline of computational fluid mechanics, while not knowing a precise application kind would be able to search for instances of interfaces derivative of a base application interface, for example a supposed **CfdAppl** interface acting as a common foundation for all computational fluid mechanics applications.

No other explicit publication of interface instances is currently anticipated by the application layer since nearly all such instances are reachable through the structures emanating from instances of the **GacAppl** interface. In particular, the application interface provides a reference to a **GOrgGObjByType** organization which provides a complete, ecdysiastical reference to all the information-bearing instances of an application.

4 Documentation

Nearly all material relevant to the PIA effort, including complete, class-by-class, member-by-member documentation, is available on a central server provided by the Glenn Research Center. The root URL for this documentation is

http://www.grc.nasa.gov/WWW/price000/index.html

It must be strongly emphasized that these pages are the generation of the researchers involved and do not in any way represent a commitment of the Government of the United States, yada, yada, yada.

5 Summary

The concepts and implementation of the Application Layer infrastructure devised for the CORBA-served, distributed object form of the Project Integration Architecture (PIA) has been presented. The following key points were discussed.

1. A network of mutually-trusting, PIA application servers, known as a **collective**, has been described.

2. The concept of a "user" in the context of a collective has been defined and the elements necessary for that user to operate and obtain useful services and resources has been described.
3. The mechanisms devised to assure not only the resolution of conflicts due to concurrent access, but the appliation and exercise of access controls to the resources made available have been described.
4. Distritbuted mechanisms for administering users throughout a collective have been described.
5. The methods for establishing and maintain trust in the correct operation of the various components of the application infrastructure has been discussed.
6. Finnally, mechanisms for locating resources wherever they might exist throughout a given collective have been described.

References

[1] The American Society of Mechanical Engineers. *Integrated CFD and Experiments Real-Time Data Acquisition Development*, number ASME 93-GT-97, 345 E. 47th St., New York, N.Y. 10017, May 1993. Presented at the International Gas Turbine and Aeroengine Congress and Exposition; Cincinnati, Ohio.

[2] James Douglas Stegeman, Richard A. Blech, Theresa Louise Benyo, and William Henry Jones. Integrated CFD and Experiments (ICE): Project Summary. Technical memorandum NASA/TM-2001-210610, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, December 2001.

[3] William Henry Jones. Project Integration Architecture: Application Architecture. Draft paper available on central PIA web site, March 1999.

[4] American Institue of Aeronautics and Astronatics. *Project Integration Architecture (PIA) and Computational Analysis Programming Interface (CAPRI) for Accesing Geometry Data from CAD Files*, number 2002-0750. Aerospace Sciences Meeting and Exhibit, Reno, NV.

[5] Theresa Louise Benyo. Project Integration Architecture (PIA) and Computational Analysis Programming Interface (CAPRI) for Accessing Geometry Data from CAD Files. Technical memorandum NASA/TM-2002-211358, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, March 2002.

[6] William Henry Jones. Project Integration Architecture: Formulation of Dimensionality in Semantic Parameters. Draft paper available on central PIA web site, March 2000.

[7] William Henry Jones. Project Integration Architecture: Distributed Lock Management, Deadlock Detection, and Set Iteration. Draft paper available on central PIA web site, April 1999.

[8] William Henry Jones. Project Integration Architecture: Inter-Application Propagation of Information. Draft paper available on central PIA web site, December 1999.

[9] William Henry Jones. Project Integration Architecture: Formulation of Semantic Parameters. Draft paper available on central PIA web site, January 2000.

[10] William Henry Jones. Project Integration Architecture: Wrapping of the Large Perturbation Inlet (LAPIN) Analysis Code. Draft paper available on central PIA web site, March 2001.

[11] William Henry Jones. Project Integration Architecture: One Possible Commercialization Plan. Draft paper available on central PIA web site, May 2002.